(4)

USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

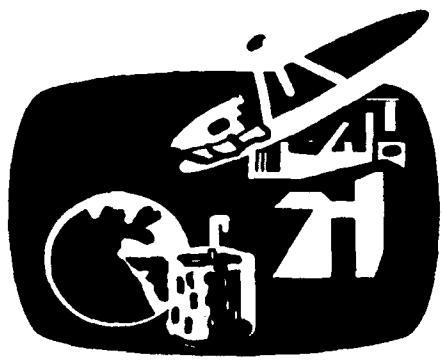
U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

A Perspective of Software Reuse

(ASQBG-I-89-025)

April 1989

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800



This research was performed as an in-house project at the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the RDTE organization of the Army's Information Systems Engineering Command (ISEC). This work was completed by Dr. Jim Hooper, University of Alabama at Huntsville, a visiting scientist under the Army's Scientific Service Program who was resident at AIRMICS from September 1988 to January 1989. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

s/ Glenn Racine

Glenn Racine, Chief
Computer and Information
Systems Division

s/ John R. Mitchell

John R. Mitchell
Director
AIRMICS

A PERSPECTIVE OF SOFTWARE REUSE

by

James W. Hooper
Professor of Computer Science
The University of Alabama in Huntsville
Huntsville, Alabama 35899

for

U.S. Army Institute for Research In Management
Information, Communications, and Computer Science
(AIRMICS)

March 1989

Contract No. DAAL03-86-D-0001
Delivery Order 1144
Scientific Services Program

TABLE OF CONTENTS

PREFACE

The work underlying this report was conducted during a period of residence at AIRMICS (September 1988 to February 1989) as part of a year of sabbatical leave. AIRMICS is very active in software reuse research, and my stay there was very pleasant and technically rewarding. I thank Glenn Racine, COR of this effort, for his cooperation and support, as well as John Mitchell, Director of AIRMICS. My thanks also to Dan Hocking for numerous worthwhile discussions on software reuse, and the other people at AIRMICS for helping me to "feel at home".

1. INTRODUCTION

In 1980 the U.S. Department of Defense (DoD) spent over $3 billion on software; by 1990, their expenses are expected to grow to $30 billion per year (Horowitz and Munson 1984). And even though expenditures are escalating, productivity is falling behind the demand for new software. The same trends are perceivable throughout the software industry--in private companies and government agencies. Jones (1984) estimates that of all the code written in 1983, probably less than 15 percent is unique, novel, and specific to individual applications. And, estimates are that on the average only about five percent of code is reused (Frakes and Nejmeh 1987, quoting DeMarco).

Thus we see an obvious candidate area for increasing productivity and reducing cost--that is, to reuse existing software products to achieve all or part of the redundant 85 percent of the development. Even a one percent gain, relative to DoD's projected $30 billion, could save $300 million! As Standish (1984) observed, "software reuse has the same advantage as theft over honest toil". In addition to increases in productivity and reduction in costs, software quality should increase due to the greater use and testing of individual components, with the resulting isolation and correction of any problems discovered.

Some software reuse has occurred for many years, of course, beginning with libraries of mathematical

subroutines, and now including operating systems, language processors, report generators, compiler generators, fourth-generation languages, and many application-specific packages. To achieve the needed benefits, however, software reuse must be expanded much further.

Both in the U.S. and abroad (especially Europe and Japan), a great deal of research is underway, seeking effective means to achieve software reuse. Some current research projects are CAMP (Common Ada Missile Packages)(Anderson and McNicholl 1985, McNicholl et al 1986), funded by the DoD STARS (Software Technology for Adaptable, Reliable Systems) program. The federally-funded Software Engineering Institute (SEI) at Carnegie-Mellon University in Pittsburgh, is conducting reuse research based on the CAMP reusable parts, as is U.S. Army CECOM's Center for Software Engineering. U.S. Army AIRMICS has a broadly-based reuse research program underway, with the support of Martin Marietta Energy Systems. A number of universities are participating in the project (entitled the Ada Reuse and Metrics Project), and the research is focusing on various facets of reuse relating to the software life cycle (Hocking 1988). The RAPID Center project is being conducted by SofTech for the U.S. Army Information Systems Engineering Command, emphasizing the identification and retrieval of reusable Ada software components (Guerrieri 1988). The U.S. Army Strategic Defense Command has funded reuse efforts--e.g., that reported by Asdjodi (1988). The Software

Productivity Consortium (Reston, Virginia) conducts reuse research (Pyster and Barnes 1987), as does the Microelectronics and Computer Technology Corporation (MCC, Austin, Texas; e.g., Biggerstaff and Richter 1987). A summary of current reuse projects, including some of those mentioned above and others, may be found in Lesslie et al 1988.

The DoD Ada Software Repository was established in 1984 to promote the exchange and use of public-domain Ada programs and tools, and to promote Ada education by providing several working examples of programs in source code form to study and modify. The Repository contains source code exceeding 20 MB in size. Conn (1986) provides an overview of the DoD Ada Software Repository, and explains how to obtain access to available services.

AdaNET is a government-sponsored information service, established in October 1988 to facilitate the transfer of federally developed software engineering and Ada technology to the private sector. It is operated by MountainNet, Inc., Dellslow, West Virginia, and sponsored by NASA Technology Utilization Division, DoD Ada Joint Program Office (AJPO), and Department of Commerce Office of Productivity, Technology and Innovation (OPTI). AdaNET offers 24 hour-per-day on-line computer access to information about Ada software, bibliographies, conferences and seminars, education and training, news events, products, reference materials and standards. Interested organizations

and individuals are invited by MountainNet to call them for detailed information, including how to apply for an AdaNET Electronic Mail Account (call (304) 296-1458)).

Significant projects are underway in the United Kingdom, as evidenced by the special section on software reuse in the September 1988 issue of the Software Engineering Journal (Hall 1988a); this issue contains some excellent research papers. A recent paper on the Japanese "software factory" approach is Fujino 1987. Many conferences and workshops dealing with software reuse are being held--they are reflected in the references in this report; and refereed journal articles on reuse are becoming more numerous. Progress is being made as will be apparent from the subsequent technical discussions in this report, and the papers appearing are becoming more substantive. There aren't yet as many practical "success stories" as we could wish, but some are occurring. Examples are the Magnavox effort reported by Carstensen (1987) (discussed briefly in section 2.2 of this report), and Raytheon's reuse efforts in business data processing applications (Lanergan and Grasso 1984); Raytheon expects a 50 percent gain in productivity with their approach.

This report makes no attempt at being all-encompassing, but rather seeks to provide a representative overview of the status of software reuse research and practice. Chapter 2 provides a technical overview; the approach is to separately

discuss various facets of software reuse, as well as their interrelationships. This includes an identification of issues, and current approaches to their resolution. Extensive use is made of the current literature in this overview. Chapter 3 provides conclusions and some recommendations for further research and for implementing available research ideas.

Software reuse is indeed a multi-faceted subject, and there are many good articles available for one seeking a further grounding in fundamentals and history. A very important seminal reference is the "landmark" September 1984 issue on software reusability of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (Biggerstaff and Perlis 1984). Other important references are the July 1987 special issue of IEEE SOFTWARE on "Making Reuse a Reality" (Tracz 1987b). An outstanding single paper that provides a good overview is Biggerstaff and Richter 1987. Additionally, many of the papers listed in the Reference section contain good overview material.

2. FACETS OF SOFTWARE REUSE

In this chapter we consider many facets of software reuse that are currently undergoing consideration and experimentation. We attempt to "decompose" software reuse in such a way as to understand the nature and issues of software reuse, to examine current research directions and practice, and to identify promising approaches and candidate areas for further emphasis.

2.1 Definitions of Reuse/Reusability

The way we define terms is critically important, since our understanding is determined (focused, limited) thereby. There seem to be two fundamental definitions of "software reusability". With variations in wording, they are:

(a) the extent to which a component can be used in a context other than the one for which it was originally developed;

(b) the extent to which a component can be used in multiple applications.

Definition (a) suggests that reuse is an incidental result from software development; definition (b) suggests planned efforts toward reuse. While the prefix "re" of the word reusability may inherently suggest definition (a), definition (b) appears to be a more productive and better-focused definition, and likely better characterizes most current research. Perhaps definition (a) better characterizes the previous achievements in reuse, and (b) recognizes the need to emphasize reuse as a worthy focus within itself (more on the latter point in section 2.7).

In this report we generally use the term "reuse" rather than "reusability"; it is a shorter word, and perhaps has the merit of conveying a more active tone than is true of "reusability". Clearly, reuse is the goal, while reusability is necessary in order to achieve the goal.

2.2 Barriers and Incentives

While software reuse is widely applauded as "a good thing", there are inherent difficulties in bringing it about in practice. Hall (1987b) cites technical, social, economic and legal barriers to reuse. The difficulties are greater between companies, and between a company and the U.S. Department of Defense (DoD), than is true within a single organization; but even within an organization, competition between groups and funding approaches inhibit reuse. Reuse is hampered by the practice of funding projects essentially in isolation--thus project A doesn't get funds to generalize software for the benefit of projects B and C, and may put itself at a disadvantage if funds are so spent.

It appears that a company must always retain some proprietary software, for competitive advantage. To the extent software is placed in a repository for general use, how is a company to be rewarded? A company must be duly compensated somehow for the expected loss of the revenue that retention of the software would have given them. A related issue is that it is so easy to steal software--and thus bypass any mechanisms devised to compensate the developers. And, what is the financial incentive to be for a contractor to reuse existing code in a DoD project (when, if they develop it "from scratch", they would make a profit in doing it)? Hall (1987b) mentions the idea of a "meter" within a software component, that counts uses, and charges accordingly. Clearly DoD and other agencies must come to

grips with the legal and contracting issues involved, for the national benefit. Tax considerations are a factor, as to how a software purchase is "written off". Software copyright laws are also an issue. Some sort of royalty and licensing fees are required, and maintenance agreements are important.

The "not invented here" (NIH) syndrome plagues us all to some extent, and has been especially pervasive among software development personnel. Programmers resist using software developed by a colleague "down the hall", let alone by another company. Healthy skepticism is a valuable asset to a software developer, in many of his/her activities. The way to overcome the NIH issue seems to be for an organization's management to strongly, consistently and intelligently emphasize and require reuse. This involves ensuring the correctness of reusable code, so that trust can be built up on the part of software personnel. It also involves effective technical mechanisms for locating and using existing code (section 2.5). Reuse achievements of personnel should result in positive rewards from management.

The primary incentive for reusing software will presumably be potential cost savings. With effective economic and legal mechanisms in place, this should be realizable. And, since cost will be spread over many projects, very rigorous verification of components should be feasible and cost-effective. A study is being undertaken by

Fairley et al (1989), to develop effective cost models for reuse. Barnes et al (1987) present a framework for analyzing the costs of reuse. The initial cost of developing reusable software is greater than that of other software, and is an inhibiting issue. Clearly the expectation must be of recovering the additional investment in subsequent uses. An additional incentive for reuse in the U.S. no doubt is the reported success in reuse of our economic competitor Japan.

An example of a successful reuse project is reported by Carstensen (1987), of Magnavox. The AFATDS project for the U.S. Army consisted of approximately 770,000 lines of Ada code, of which about 100,000 lines were reused code. Of the 100,000 lines, about 30,000 were reused unchanged, and about 70,000 resulted from tailoring existing modules. They used object-oriented design (section 2.7) and believe it facilitated reuse. By way of incentive for reuse, at project initiation they determined and costed a specific software reuse factor, that had to be met to stay on schedule and within budget. As Carstensen notes, this requires acceptance of some risk by project-level management; and he emphasizes that whatever incentives are used, the single most important incentive (factor) is the acceptance by project management of any real or perceived risks associated with the reuse of previously developed software. Lanergan and Grasso (1984) also emphasize the role of management personnel in Raytheon's successful reuse

project.

There are numerous remaining issues, including lack of methodologies incorporating reuse, inadequate library mechanisms, and the use of many different programming languages. But, as Bott and Wallis (1988) note, "the main obstacles to the widespread adoption of reuse as a standard way of writing software are managerial rather than technical." And, they further observe that "Software reuse will not happen merely because the technical means of achieving it become available. Nor will it be applied successfully within an organization merely because it becomes official policy."

There are efforts underway to establish guidelines for reuse within organizations. Examples are Lesslie et al 1988, SofTech 1985, and St. Dennis 1987. And there is at least one commercially-available reuse library--offered by EVB Software Engineering, Inc. (Reston, Virginia), based on Booch's work (Booch 1987). The AdaNET project, summarized in Chapter 1, can become a good reuse sharing mechanism, and perhaps a good "testbed" for dealing with some of the issues mentioned above.

For the benefit of those who may wish to pursue these issues further, good discussions may be found in Biggerstaff and Richter 1987, Hall 1987b, Geary 1988, Bott and Wallis 1988, Wood and Sommerville 1988, Frakes and Nejmeh 1987.

2.3 Abstraction Levels of Reuse

The "products" for reuse may be considered and characterized relative to activities of the life cycle. We can characterize these activities as:

* domain analysis
* requirements specification
* high-level design
* detailed design
* coding
* testing and integration
* documentation
* maintenance

In principal, we should expect a greater return from the higher-level abstraction activities, if their products can be reused. Thus, a reused requirements specification should give us greater leverage than a reused code module. Much knowledge is gained from the life cycle activities, only part of which is usually recorded and retained. In addition to requirements specifications, designs, code, test documents, test cases, integration plan, etc.--which are recorded, we also have learned much about the application domain during a project, and have determined rationale for design decisions, tradeoff considerations in decomposition and allocation to system components, etc. And almost always some knowledge is "factored out" as we proceed through the refinement process; but knowledge--acquired "the hard way" during a project--is exceedingly valuable for retention.

Some of the research efforts summarized in this report are focusing on this issue, and are devising means to retain knowledge and experience for subsequent use. Examples of the approaches are entity-relationship-attribute models, and rule-based approaches. Yamamoto and Isoda (1986) explicitly address this issue, and propose a mechanism for capturing the experience during a project. Ramamoorthy et al (1986) propose a means to recapture some information from existing code--recognizing that much of the needed information is not available in the code (they recommend structured comments for provision of additional information). Bott and Wallis (1988) and Biggerstaff and Richter (1987) emphasize the need for a simplified "user model" (or, "mental model") of a system, to aid reuse. More about these ideas in subsequent sections.

It should be noted that experienced personnel naturally retain much knowledge of their previous work; thus "reuse of personnel" is extremely advantageous in software development within an application area. It appears that much of the productivity increase in software reuse reported by Japanese software developers is due to their success in retaining experienced personnel, and capitalizing on the knowledge and experience they have gained. Fujino (1987) comments on the balance of emphasis between motivation of personnel, and software automation.

Mechanisms for reusing software products other than

code have been slow to appear, but a number of research efforts now underway are beginning to address this need. An example is the work of Finkelstein (1988), who is conducting research in means to detect opportunities for reuse based on requirements specifications. He made an attempt to "marry" his concepts with an existing requirements specification language with little success, and concludes that a requirements language should be devised in which the primitive elements have been selected with reuse in mind. Neighbors (1984) has developed the Draco approach, in which he proposes the use of a different "domain language" for each different problem area. The objects and operations of a domain language would represent analysis information in the problem domain; thus analysis and design information, in addition to available code modules, would be reused each time the domain language is used to describe a problem solution.

In this report, the word "component" is used to mean any type of software entity that may be reused (e.g., code modules, designs, requirements specifications). In the next section we consider various approaches to the reuse of components of source code.

2.4 Mechanisms for Reusable Code

Reusable code is the most-researched of the reuse abstraction levels; more is known about it, more has been written about it, and indeed most of the software reuse has been based on code alone.

There are two different clearly-distinguishable categories of code components for reuse. The first may be called "passive" components, or "building blocks", which are used essentially unchanged. Another approach--very effective when feasible--is "dynamic components" (i.e., "generators"), which generate a product for reuse. This approach is also spoken of as reusable "patterns". Generators are of two fundamental types (Biggerstaff and Richter 1987): (a) application generators (employing reusable patterns of code), and (b) transformation systems (which generate a product by successive application of transformation rules; e.g., see Cheatham 1984). Generators by their nature tend to lift the abstraction level for reuse above that of code building blocks. For example, compiler syntax analyzer generators (e.g., YACC) do their work with little need for the user to understand the underlying concepts. Simulation languages/systems constitute another application of reusable patterns to achieve effective leverage (Hooper 1988). Prototyping usually is based on significant reuse of software; it may be very-high-level-language based, somewhat like simulation languages, and/or may be based on code blocks. Fourth generation languages

constitute another kind of reusable pattern, and also substantially lift the reuse abstraction level.

Code blocks for reuse have historically been subprograms (procedures, functions), and now also include Ada packages (collections of reusable subprograms with their encapsulated environment), classes (in the sense of object-oriented programming), and Ada generics. Parameterized code may be based on any of the forms of code blocks just mentioned. Also, code templates may be used, with slots to be filled in by a user to customize for a given application.

Reference to these different ideas for reusable code components will occur in the subsequent sections.

The recent past has brought better programming languages for reusability support--especially Ada. Modula-2 and various other available languages also have some good features. But the DoD mandate for use of Ada doubtless makes Ada the language of choice for code reuse, and Ada has very strong features, such as the package (information hiding, encapsulation) and generics. More on this in later sections.

2.5 Mechanisms for Reuse Repository Operation

There are a number of operational issues that must be addressed in order to effectively make use of available reusable software products. Included are the need to (a) classify and retain components ready for reuse; (b) find components that meet specific needs; (c) understand identified components; (d) adapt components as necessary; and (e) compose components into a complete software system. These operational issues span the abstraction levels of software realization (section 2.3)--i.e., they apply to components of requirements specification or design just as they do to code. However, to the present time research and experience have primarily pertained to code modules.

In the following subsections we consider the five operational issues (a) through (e) stated above, in turn.

2.5.1 Classifying and Retaining Components

Considerable research activity is underway in this aspect of reuse repository operation. Prieto-Diaz and Freeman (1987) have developed the "faceted" classification method, based on ideas from library science. Each component is characterized by a sextuple consisting of:

<function, objects, medium, system type,
functional area, setting>.

Figure 1 (from Prieto-Diaz and Freeman 1987) gives an idea of what each of the facets means. They incorporate the idea of "conceptual closeness", to give a user a measure of how

closely an available component corresponds to a specified facet during retrieval.

Function	Objects	Medium	System type	Functional area	Setting
add	arguments	array	assembler	accounts payable	advertising
append	arrays	buffer	code generation	accounts receivable	appliance repair
close	backspaces	cards	code optimization	analysis structural	appliance store
compare	blanks	disk	compiler	auditing	association
complement	buffers	file	DB management	batch job control	auto repair
compress	characters	keyboard	expression evaluator	billing	barbershop
create	descriptors	line	file handler	bookkeeping	broadcast station
decode	digits	list	hierarchical DB	budgeting	cable station
delete	directories	mouse	hybrid DB	capacity planning	car dealer
divide	expressions	printer	interpreter	CAD	catalog sales
evaluate	files	screen	lexical analyzer	cost accounting	cemetery
exchange	functions	sensor	line editor	cost control	circulation
expand	instructions	stack	network DB	customer information	classified ads
format	integers	table	pattern matcher	DB analysis	cleaning
input	lines	tape	predictive parsing	DB design	clothing store
insert	lists	tree	relational DB	DB management	composition
join	macros	.	retriever	.	computer store
measure	pages	.	scheduler	.	.
modify	.	.	.	.	.
move	.	.	.	.	.
.	.	.	.	.	.

Figure 1. Sample Facet Choices
(from Prieto-Diaz and Freeman 1987)

Ramamoorthy et al (1986) have selected the entity-relationship-attribute (ERA) model as the basis for cataloguing and retrieving reusable components. They use such attributes as classification (requirements, design, source, test case, document, library, object code), hierarchy (family, member, layer, module, procedure), and non-functional attributes such as reliability, memory requirements, performance, and metrics concerning quality and complexity. Their relations include classification-to-

classification, hierarchical_level-to-hierarchical_level, use of resources, etc. They have developed the Entity Specification Language (ESL) to support software personnel in inserting, modifying or deleting information in the library.

Mittermeir and Rossak (1987) have proposed library organizations which they call "software archives" (to support the retrieval of design units for potential reuse), and "software bases" (to support retrieval of executable code). They represent the structure of a software archive by a four-dimensional "cube", the dimensions being decomposition, representational form, association (application-dependent links), and generalization /specialization. This structure thus uses links among components to represent knowledge about interrelationships.

Wood and Sommerville (1988) have taken a cataloguing approach based on natural language processing--specifically the idea of "concept case frames". They rely on a system creating component descriptor frames (their version of concept case frames) by means of a form-filling interface. There is a component descriptor frame for each basic function that software performs--representing a class of conceptually-similar verbs (e.g., search, look, and find). There are four slots in each frame corresponding to: a library component, objects manipulated by the component, objects produced as a result of the action, and objects that

provide a context for the action. An example is (in order): a report generator, a personnel record, a formatted report, directives that describe the desired report. This organization was devised to improve the process of retrieval (subsection 2.5.2).

The RAPID Center Project was mentioned previously. As part of this project (being pursued by SofTech under contract to the U.S. Army Information Systems Engineering Command), the RAPID Center Library (RCL) System is being developed (Guerrieri 1988). Classification is based on the faceted classification scheme of Prieto-Diaz and Freeman (1987). User activities supported relative to reusable software components are identification, extraction, and report generation. The RAPID Center will also provide guidance to users by technical staff.

Gagliano et al (1988) have developed an experimental reuse library system, based in part on Prieto-Diaz's work, for reusable Ada components. They plan to develop several tools to ease the development and use of a reuse library.

All the kinds of products discussed in section 2.3 should be retained in the repository. And, there should be relationship information between components (e.g., between a design component and the corresponding code, between a system specification and a subsystem specification, between two components that are related by reuse potential within a given application domain, and between two components such that one is a specialization of the other).

Research is underway to examine appropriate structures for database support for software reuse (e.g., Bein et al, 1988; Frakes and Nejmeh 1987)--i.e., to determine how best to use the knowledge and systems of database management and information retrieval to support the creation and use of software repositories.

2.5.2 Searching and Retrieving

In the case of the faceted classification scheme (Prieto-Diaz and Freeman 1987), the search is based on specification of a sextuple of descriptive keywords (see 2.5.1). The user forms a query using the thesaurus to select a representative term for each facet. The user can use the asterisk rather than a keyword for a facet, to generalize the query. The user can also request "expansion" of a query by having one of the specified facets replaced by other keywords, in order of conceptual closeness. The researchers have also attempted to provide ranking of the retrieved components based on user profiles; e.g., it is more important for novice programmers to attempt to use very small programs (in terms of lines of code) than is true for more experienced programmers.

Ramamoorthy et al (1986) have developed the Resource Extractor (REX) for use in forming queries using attributes and/or relations as qualifiers. Example queries (from Ramamoorthy et al 1986) are:

```
SET High_Req = Software_Resources
    Classification = Requirements
    Performance = High.
```

This query results in the creation of a set High_Req containing those elements of the set Software_Resources whose elements satisfy the specified relation and attribute.

Wood and Sommerville (1988) provide a forms-based interface to their library implementation on a Sun workstation; it prompts the user for either a verb describing the action of a component or for a noun representing an object manipulated by the component. Thereby the system undertakes construction of a component descriptor frame (section 2.5.1); partially-completed frames are used to search the database; the user is provided lists of candidates for filling in other slots, and on-line helps are available. The user can select either a keyword search, in which an exact match on specified names must occur, or can permit components to be retrieved based on conceptual classes of verbs, for example. They plan to extend the system to include a browsing mode, and tools to integrate reuse with automated design tools.

Tarumi et al (1988) have developed a rule-based retrieval mechanism, based on user inputs of object names, attributes, relations, and operations. Names may have aliases. Their approach is a mixture of formal and informal methods; they emphasize the importance of this combination, believing that it yields simplicity and user friendliness.

Frakes and Nejmeh (1987) propose the use of information retrieval systems for locating code components for reuse (and this could be extended to other abstraction levels). Information retrieval (IR) systems deal with formatted text, as well as unformatted text which is not usually dealt with by database management systems. They have devised a novice user mode, which is menu driven, and a command mode for more experienced users. They have developed an experimental system called CATALOG, written in C under UNIX and MS-DOS. Help messages are available, and partial term matching is done using such sophisticated IR techniques as automatic stemming and phonetic matching. They propose that all components submitted to a reuse library must begin with a standard prologue of descriptive information, to form the basis for subsequent search and retrieval of the component (e.g., name, description, supporting documents, author, date, usage, parameters). With the advent of special-purpose hardware for IR (e.g., see Smith 1989) and resulting greatly-improved responsiveness, IR techniques are worthy of research emphasis for reuse support. Frakes and Nejmeh (1987) also mention the promise of user feedback in conjunction with IR use.

Three important aspects of the retrieval process are discussed by Wood and Sommerville (1988), which are concepts from information retrieval. They are recall, precision, and ranking. Recall pertains to the percentage of relevant

components that are identified; precision pertains to the percentage of identified components that are relevant; and ranking orders identified components by quality of match (to address the "information overload" issue). These are all substantive issues, and will be increasingly important as reuse libraries become larger.

In section 2.2, we summarized a successful research project reported by Carstensen (1987). He described an approach they took to library browsing, in which the user may specify English language nouns and verbs as search vectors. Based on potential matches, the user is first presented two-sentence abstracts of all candidate components. If desired, a more complete abstract (up to one page in length) may be requested for any of the components.

2.5.3 Understanding Identified Components

Standish (1984) estimates that software maintenance costs 70-90 percent of the life cycle, and understanding accounts for 50-90 percent of maintenance cost. This would mean that understanding accounts for 35-80 percent of life cycle cost. Understanding is absolutely critical to software reuse--especially if a component must be adapted.

Berard (EVB 1987) discusses the importance of good software engineering practice for reuse. Coding style is very important--e.g., using meaningful identifiers, avoiding literal constants, using adequate, concise and precise

comments, making frequent and appropriate use of packages, isolating and clearly identifying environmentally-dependent code; and, modules should be highly cohesive and loosely coupled. In general, as implied above, the approaches that promote understanding for maintainability serve equally well for reuse. Indeed, Barsotti and Wilkinson (1987) argue that reusability of code is essentially a by-product of quality and maintainability. To this end, they recommend:

* using certified algorithms
* defining error handling
* defining exceptional conditions
* establishing explicit interfaces
* developing modular programs
* parameterizing
* providing all test data and reports

They also provide a list of "quality criteria", as follows: accuracy, application independence, augmentability, completeness, conciseness, consistency, fault tolerance, generality, legibility, self-descriptiveness, simplicity, structuredness, system accessibility, and traceability. They provide precise definitions to all these terms. By application independence they mean nondependency on the database system, microcode, computer architecture and algorithms (i.e., not independence from an application domain); this thus includes the important criterion of portability. By system accessibility, they mean provision

for control and audit of access to the software and data.

They indicate that all these criteria support reusability; all support maintainability except accuracy, application independence and fault tolerance; and reliability is supported by all except application independence, augmentability, generality, self-descriptiveness, simplicity, and system accessibility. While one could argue with some of these particulars, the concepts are worthwhile. These and other criteria relate to the concept of software metrics--criteria by which the suitability of software may be measured.

St. Dennis (1987) suggests a list of 15 language-independent characteristics of reusable software, as follows: (1) interface is both syntactically and semantically clear; (2) interface is written at appropriate (abstract) level; (3) component does not interfere with its environment; (4) component is designed as object-oriented; that is, packaged as typed data with procedures and functions which act on that data; (5) actions based on function results are made at the next level up; (6) component incorporates scaffolding for use during "building phase"; (7) separate the information needed to use software, its specification, from the details of its implementation, its body; (8) component exhibits high cohesion/low coupling; (9) component and interface are written to be readable by persons other than the author; (10) component is written with the right balance between generality and specificity;

(11) component is accompanied by sufficient documentation to make it findable; (12) component can be used without change or with only minor modifications; (13) insulate a component from host/target dependencies and assumptions about its environment; isolate a component from format and content of information passed through it which it does not use; (14) component is standardized in the areas of invoking, controlling, terminating its function, error-handling, communication and structure; (15) components should be written to exploit domain of applicability; components should constitute the right abstraction and modularity for the application.

In addition to these language-independent guidelines, St. Dennis provides seven Ada-specific guidelines, which will not be repeated here.

Bott and Wallis (1988) argue that (a) we need to use components that implement fairly complicated functions in order to achieve large benefits from reuse, and that (b) in order to do so it is essential to reduce the perceived complexity of components as seen by the system designer. They maintain that, to this end, components must be designed for reuse from the beginning; and, their major theme is that components must conform to some kind of simplified "user model" of the system which they support, to relieve a user of detailed coding concerns. An example of such a user model is that of compiler construction (the front-end/back-

end division, and the compiler phases). They also note that UNIX users benefit from the simplicity of its "user model"; although it is a very complicated system, enough can be learned very quickly to allow useful activities to occur. Biggerstaff and Richter (1987) refer to this simplified model as the "mental model", and state that developing such a model is probably the fundamental operational problem to solve in development of any reuse system. They suggest the use of hypertext to help solve the problem.

Chen and Ramamoorthy (1986) have developed the C Information Abstractor, which scans C programs and stores information into a database. The information obtained primarily relates to objects that can be accessed across C file or function boundaries--namely, files, functions, global variables, global types, and macros. An Information Viewer is provided to operate on the database, and provide answers to such questions as: what functions call a given function, where is a certain variable defined, what functions access a given global variable, what is the type of a variable. Since some important information cannot be extracted from the code (e.g., underlying assumptions, the algorithm used, computational complexity, necessary preconditions), the authors propose the use of structured comments to provide the information to the Abstractor. Examples of such information they suggest are: purpose, assumptions, preconditions, assertions, algorithm description, algorithm complexity.

Chen and Ramamoorthy (1986) also comment on software metrics, noting that software quality, testing required, and maintainability, depend on such metrics as function-to-file bindings, file-to-file bindings, the number of objects related to a given function, and the number and depth of calling paths starting from a function. They observe that an examination of such metrics may well indicate the need for restructuring prior to reuse. They are considering means to handle some of the details automatically, using the program database.

Basili et al (1988) are undertaking research to evaluate the reuse implications of Ada modules based on the explicit and implicit bindings of the module with its environment. They are planning tool support, and have formulated guidelines for code development based on the research conducted so far.

Research in "reverse engineering"--approaches for "unraveling the product ... to its earlier life cycle development phase(s)" (Sayani 1987), can provide important leverage in aiding the understanding process, and alleviating to some extent the need to attempt to understand code.

2.5.4 Adapting Components

The ideal situation is that a component (or components) will be identified which exactly meets the need. That will

doubtless often not be the case, however. Understanding the component (subsection 2.5.3) is the key to the decision process, of course. "Goodness of fit" of an available component might well be measured by the effort required for adaptation. This would provide guidance when multiple components are identified which are candidates for selection. If a component is functionally adequate--i.e., it performs a needed role in an acceptable way, then there should be little or no adaptation required if the component is highly cohesive and has no side effects.

Ramamoorthy et al (1986) give some parameters for use in deciding whether to reuse a component. They are:

Nnr: Number of lines to be written if no reuse;

Nmr: Number of lines to be modified for reuse;

Nr: Number of lines which are being reused;

Enl: Effort the organization needs for writing a new line;

Eml: Effort the organization predicts for modifying a line;

Emal: Effort the organization needs for maintaining a line;

Nrb: Number of times previously reused;

Net: Number of errors found in some fixed period;

Nel: Number of errors found per fixed number of lines;

DocQ: Documentation quality;

DesQ: Design quality;

M: Match for non-functional characteristics;

Exp: Experience with the package to be reused;

Org: Organization maintaining the package to be reused;

Dev: Availability of the original developers;

Time: Time available for the project.

They suggest some basic rules for deciding which component to reuse, such as:

(1) Nmr * Eml should be less than Nnr * Enl;

(2) Net and Nel should be less than some predefined constant;

(3) DocQ and DesQ should be greater than some predefined minimum constants;

(4) If Nr > CNr and Nrb > CNrb, then reuse the component (where CNr and CNrb are appropriately defined constants). This is relying on the likelihood that a more frequently used component is of higher quality for reuse.

If code requires adaptation, the design and/or specifications corresponding to the code component (and hopefully retained in the repository) will likely prove to be very important. If the component must be rewritten in a different programming language, the high-level design should serve as the basis--which is also true if the number of code "patches" required for adaptation is large.

Parameterized code is developed with the intent that input parameters cause "adaptation" of the code, as pre-planned. And, generators are driven by input directives to

"adapt" within a pre-planned range. Ada generic procedures provide a mechanism for developing a "family" of procedures for which data types may be specified--and thus a specific "adaptation" created.

Asdjodi (1988) has developed, as part of a prototype reuse system (discussed further in section 2.7), the capability to automatically alter data structures as required for use by a selected component; thus, for example, if the output of one component is a matrix, and the input for another is a linked list, her knowledge-based system would cause automatic generation of the required linked list. In the prototype system only matrices and linked lists are used; however, the concepts could be extended to any type of data structures.

Notkin and Griswold (1988) have developed a UNIX-based "extension" mechanism, based on an Extension Interpreter (EI). The EI consists of an arbitrator, which hierarchically maps procedure names to procedure implementations; the dynamic linker, which gives the flavor of interpretive environments like LISP; and the translation subsystem, which translates data between representations used by different languages. These components connect program components with a user interface. Their emphasis is on reusing source code without the need to change it. Thus, the more fine-grained the available procedures, the more likely that new capabilities can make use of them.

2.5.5 Composing Components

Composition refers to interconnecting components to form software systems. Numerous ideas have been proposed; obviously the most straightforward approach is when the component is a procedure which perfectly meets the need; then composition results from a procedure call, coupled with the action of the "linker". The same is true, of course, if we can successfully adapt a procedure for reuse. This is a very convenient process, but it is very limiting as the only mechanism for composition.

Goguen (1986) proposes to achieve composition by means of the "library interconnection language" (LIL). As it stands, LIL's syntax is Ada-like, and relies on Goguen's earlier work based on specification by use of axioms. He lists the following desirable techniques for constructing new entities from old ones:

(1) set a constant (e.g., the maximum depth of a stack);

(2) substitute one entity for a stub or parameter in another;

(3) sew together two entities along a common interface;

(4) instantiate the parameters of a generic entity;

(5) enrich an existing entity with some new features;

(6) hide (abstract or encapsulate) some features of an existing entity, either data or control abstraction;

(7) slice an entity to eliminate unwanted functionality; or

(8) implement one abstract entity using features provided by others (leading to the notion of a vertical hierarchy of entities).

LIL is an example of a "module interconnection language". The goal of such languages is to interconnect modules--which may be written in different programming languages--without having to modify the modules, assuming that they provided needed functionality.

In section 2.7 the object-oriented MELD mechanism of Kaiser and Garlan (1987) is discussed. The idea is to compose object-oriented components by merging data structures and methods from different components.

Tracz (1987a) describes a reuse system based on Ada components, using both parameterization and application generators. He describes an interactive dialog of menus and prompts to obtain necessary parameters for a particular application. Then, based on the component library and the parameters, the generator creates a compilable Ada application program. In order to prepare for reuse in a given application domain, it is necessary to do a "domain analysis" (discussed in section 2.6), identifying likely candidate applications. Parameterizing must be done, ranging from something as simple as character strings that may be substituted in the source code, to specification of how to assemble pieces of a program. In the latter case, it could be that an existing program was "dissected" for just this purpose, as a result of domain analysis.

Other important mechanisms for composition are UNIX pipes, and inheritance in object-oriented languages. Both of these have considerable benefit in shielding the user from the need to understand code, per se; in the best case the code can be treated as a "black box".

2.6 Identifying Reusable Components

As we have noted in section 2.2, it is necessary to systematically undertake software reuse within an organization if it is to be effective, which involves committment by an organization's management, including funding to "make it happen". Necessary to success is a well-populated repository of software components, along with mechanisms for repository management (the latter is discussed in section 2.5).

A few organizations have used the approach of taking all the software being developed and placing it into the repository (referred to by one wag as a "software junkyard"). The software components should be carefully chosen, considering that often the development costs should be considerably higher than for the usual software. This may be due to the form the reusable software takes (generic, parameterized, application generator, etc.), and to the rigor employed in testing it, due to its expected repetitive use.

There seem to be basically two categories of software that are good candidates for reuse. These could be referred to as horizontally-reusable and vertically-reusable components. Horizontal reuse refers to reuse across a broad range of application areas (such as data structures, sorting algorithms, user interface mechanisms), while vertical reuse refers to components of software within a given application area that can be reused in similar applications within the

same problem domain (Tracz 1987a). Horizontal reuse has, no doubt, been studied the most so far--e.g., Booch's work (Booch 1987), and likely such reuse has occurred much more frequently than vertical reuse. The main reasons for this likely are that such reuse is better understood and easier to achieve. On the other hand, the greatest potential leverage can come from vertical reuse--by intensive reuse of carefully crafted solutions to problems within an application domain. The CAMP project (McNicholl et al 1986) is an example of vertical reuse.

In order to achieve vertical reuse, a "domain analysis" is required--meaning that the processes and objects which make up the domain, and the relationships between them, are understood and recorded (Hutchinson and Hindley 1988). Hutchinson and Hindley (1988) report on their work in developing a domain analysis method. Their goals were:

* to discover the functions that underwrite reusability;
* to focus the domain specialist's attention on reuse;
* to help the domain specialist ascertain reuse parameters;
* to discover how to redesign existing components for reuse;
* to organize any domain for reuse.

The domain analysis was done by a "reuse analyst" with the assistance of a "domain specialist"--an individual with an excellent understanding of the problem domain. The researchers developed "structured domain analysis

techniques" based on questions devised to assess a software component's reusability. The domain on which they based their experimentation was a simulation of the utility systems management (USM) system of the Experimental Aircraft Programme (EAP) in the United Kingdom. The subdomains they considered were propulsion, fuel management, and undercarriage. These were chosen by the domain specialist for reuse consideration due to the fact that, in the case of propulsion, the controlled hardware (the engines) would not change significantly between the EAP implementation and the next project; fuel management was chosen because the domain appeared to contain a lot of functional duplication within the requirements definition; undercarriage was chosen because much of its operation would not change on future implementations.

The reuse analyst decided on three levels of reuse, to clarify the domain: the initial level pertained to reuse of the whole system, the next level to reuse of subsystems, and the final level to functions at the requirements level and to components at the design and code levels. The reuse analyst presented twelve questions to the domain specialist, based on the assumption that it is domain-specific knowledge which can isolate reusable components. The questions seek to elicit identification of reuse attributes and reusable components in an understandable manner. The questions are:

* Is component functionality required on future implementations?

* How common is the component's function within the domain?

* Is there duplication of the component's function within the domain?

* Is the component hardware-dependent?

* Does the hardware remain unchanged between implementations?

* Can the hardware specifics be removed to another component?

* Is the design optimized enough for the next implementation?

* Can we parameterize a non-reusable component so that it becomes reusable?

* Is the component reusable in many implementations with only minor changes?

* Is reuse through modification feasible?

* Can a non-reusable component be decomposed to yield reusable components?

* How valid is component decomposition for reuse?

The analysis resulted in the following:

In the propulsion subdomain, of 19 functional requirements identified, one component was classified as reusable without change, 14 were classed as reusable with slight modification, and four were classed as non-reusable.

In the fuel management subdomain, of 26 functional requirement components identified, two were classed as reusable without change, 14 as reusable with slight modifications, and 10 as non-reusable.

In the undercarriage subdomain, of 10 functional

requirements components identified, all 10 were classed as reusable with slight modifications from the requirements level.

The authors observe that reuse proved to be practical, even in the hardware-dependent areas being analyzed. They assessed the requirements functions as potentially 75% reusable for the next implementation, and indicated that reuse could be equally high for code designed for reuse from these requirements.

They describe their work in preparing one reusable software component in Ada for the fuel management subdomain. This was a valve control simulation--significant because the requirement for a valve control simulation was repeated in at least 11 places. In preparing the Ada code, they emphasized Ada strengths for reuse--namely, information hiding, data encapsulation, packages and generics. They first devised a package based on the original design, then generalized the package to handle similar valves by creating a valve type to hold status information on the valve. Then they abstracted the problem to yield a general package to operate a two-state device which takes a finite amount of time to switch between states; this resulted in a generic Ada package.

They report that producing the generic package took approximately five times as much development time as coding the original software design (in PDL). Most of this time

was spent educating the Ada designer in the requirements for a truly generic module. They expect that once a designer understands Ada generics and where they are applicable, the development overhead would be about twice that for a non-generic module.

The above rather lengthy description of the work of Hutchinson and Hindley has been provided because it is thought to offer very good insights into the problems of working through the process of identifying potentially reusable components.

Tracz (1987a) also goes through an example of domain analysis, and the subsequent reusable software design, based on his use of both parameterization and application generators. His general approach to software composition is summarized in subsection 2.5.5.

Prieto-Diaz (1987) gives a good discussion of domain analysis, including examples from Raytheon (business applications; Lanergan and Grasso 1984), and the CAMP project (McNicholl et al 1986).

2.7 Software Development Incorporating Reuse

It is apparent that software reuse will not occur without an organized plan to bring it about. Also, reuse requires (and deserves) integration into the software development process. At present, as stated by Ramamoorthy et al (1986), "reusability is a matter of coincidence rather than the driving force behind the software development". Based on research results, Finkelstein (1988) states, "Reuse cannot be tacked on to existing software engineering techniques, but must be built in at conception." This implies significant alteration of the software development process and supporting tools.

An increasingly-important methodology is object-oriented design (OOD). It is considered by many researchers to be promising relative to software reuse. Booch (1987) combines object-oriented design with component reuse--and has spurred a great deal of interest in the promise of reuse. Another influential advocate of OOD as the basis for reuse is Meyer (1987). His Eiffel language serves as the basis for his reuse research and recommendations.

Lieberherr and Riel (1988) have designed the Demeter system based on OOD, coupled with parameterized classes. They seek to "grow" software (as recommended by Brooks (1987)) through inheritance and parameterization, rather than building software directly "from scratch".

Kaiser and Garlan (1987) have sought to improve OOD for reuse by devising a notation (called MELD) which is

independent of any object-oriented language (and would be translated into a conventional programming language). Their system supports composition of components through merging of data structures and methods from two or more "features" (their name for reusable building blocks--similar in concept to Ada packages). They employ inheritance and data structure/behavior encapsulation from OOD.

Rogerson and Bailin (1987) conducted an experiment in reuse based on OOD versus functional decomposition, determining that is is easier to detect reusability within a given context for objects (which they represented as Ada packages).

As was mentioned above, Brooks (1987) has recommended a fundamental change to the software development process. He suggests the use of high-level prototyping, with successive refinement into code. Yeh and his associates have contributed research to life-cycle approaches incorporating prototyping (e.g., Yeh and Welch 1987).

Notkin and Griswold (1988) suggest a fundamental alteration to the software development process, by means of an "extension" mechanism (supporting the incremental extension of existing software (procedures)), to achieve the enhancement of software (which accounts for nearly 40% of the total life-cycle costs of software development). They argue that their extension mechanism can improve the overall software development process by encouraging bottom-up

development and testing, as well as enhancement.

Ramamoorthy et al (1986) propose a reusability-driven methodology, as shown in Figure 2. They emphasize the need for supporting tools, as discussed in section 2.5 of this report.

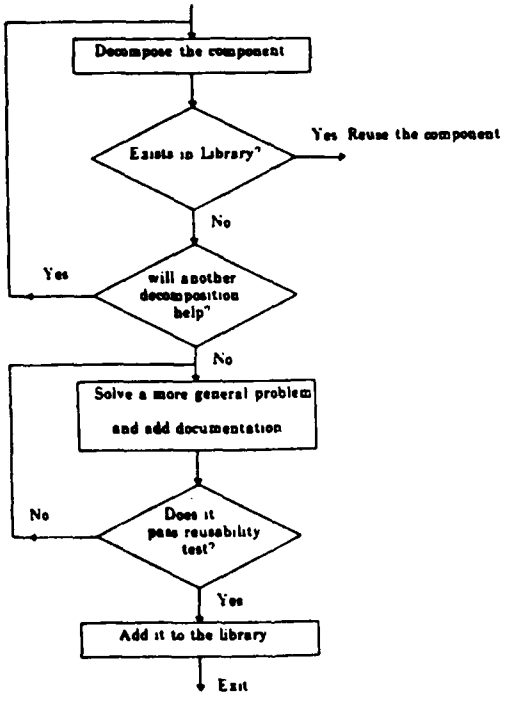


Figure 2. Reusability-Driven Development Methodology (from Ramamoorthy et al 1986)

Ramamoorthy's suggested approach is similar to that of Asdjodi (1988). Asdjodi has developed and prototyped a system based on the application domain of graph theory, incorporating domain and programming language (Modula-2)

knowledge, along with a very high level language, to support the alteration and composition of library components. Her approach also emphasizes reuse as an integral activity of the software life cycle.

Tarumi et al (1988) have developed a programming environment for object-oriented programming, supporting reuse of classes, with emphasis on an expert-system-based retrieval mechanism. McKay (1988) has undertaken to develop a life-cycle approach incorporating reuse, based on the object-oriented method.

Burton et al (1987) of Intermetrics have conducted research which seeks to provide tool support to integrate reuse within the software life cycle; their system is called the Reusable Software Library (RSL). Their approach makes use of a graphic interface (SoftCAD) for high-level design expression and automatic documentation generation, coupled with Ada PDL for detailed designs. The Ada PDL is automatically scanned for structured comments to determine candidates for reuse. The user submits queries for needed components; the researchers are experimenting with both menu-driven and natural-language based approaches for expressing queries. They attempt to help a user evaluate candidate components by rating the candidates against specified functional and qualitative attributes.

The current emphasis on CASE (Computer-Assisted Software Engineering) tools is a very strong trend. While some of the vendors list reuse support as a benefit, there

seems to be none which promote and support a specific methodology that incorporates reuse. The available capability seems to be to catalog components under a given name, and then retrieve by means of that name.

An important additional need during the software development process is to identify potentially reusable components for addition to the reuse repository. This is discussed in section 2.6.

Fischer (1987) notes H. A. Simon's observation that the evolution of a system proceeds much faster if stable intermediate parts exist. The idea is that these software parts would be integral to the design process, just as hardware IC's are to hardware design. Geary (1988) has observed: "At present, software design is created and components are only used if their specifications match requirements. Hardware design, on the other hand, is created to take full advantage of available components. In summary, hardware is designed to use components, whereas software uses components only if they suit the design."

A different mind-set, real committment to reuse, and alteration of "business as usual" in software development are necessary in order to realize the benefit of reusing software.

3. CONCLUSIONS AND RECOMMENDATIONS

3.1 Conclusions

Considering how very recently software reuse has been directly addressed by researchers and practitioners, a great deal of progress has been made. For many years incidental reuse has occurred, and planned reuse has occurred with such software as operating systems, language processors, subroutine libraries, compiler generators, report generators, graphics packages, accounting packages, and spread sheets and many other fourth-generation languages. These incidences of planned reuse have given an excellent return on investment. Much more is needed, however, in view of the continuing (and growing) cost overruns, late deliveries, and error-prone and difficult-to-maintain delivered software.

Many organizations, both private and government, are conducting software reuse research and experimentation in many countries, including the U.S., Japan, and several European nations. The quality of the research being reported has taken a marked upward turn, with very promising results. The number of practical success stories isn't as large as one could wish, but successes are occurring. Notable successes are Magnavox's AFATDS project for the U.S. Air Force (Carstensen 1987), and Raytheon's business applications reuse (Lanergan and Grasso 1984). The CAMP project (McNicholl et al 1986) is spurring a great deal of

experimentation, which should prove very fruitful. Also, the U.S. Army's RAPID Center project should give needed experience with large-scale reuse. The AdaNET operation offers a good medium for reuse communication and dissemination; the DoD Ada Software Repository also provides a sharing mechanism for public-domain software. AIRMICS' continuing reuse project is addressing various interrelated issues of reuse. The major software research initiatives in the U.S. are contributing also to reuse research (MCC, Software Productivity Consortium, SEI, STARS, etc.). Computer science departments and software engineering programs are involving graduate students in both sponsored and non-sponsored software reuse research. Also, national workshops are focusing on software reuse, and sessions on reuse are being held in many international conferences--including software engineering conferences as well as those emphasizing application areas, such as simulation and telephony/telegraphy. Ada has become available, providing effective language constructs for reuse. There appears to be sufficient momentum at this time to achieve significant advances in software reuse.

There are some very notable lacks/needs as yet, however. The major successes so far have been in specific application areas--i.e., vertical reuse (e.g., the Magnavox and Raytheon efforts), while research results have primarily emphasized horizontal reuse. Clearly the highest payoff

will tend to come from vertical reuse, since the software systems involved tend to be very large and complex, and often have a great amount of redundancy. Thus the CAMP project, and efforts based on the CAMP reusable missile parts, are important research efforts.

The repository management issues have received much emphasis, and high-quality research and experimentation have occurred. But there is as yet apparently no commercial product on the market for managing libraries of components (other than general database management systems). There is no full life-cycle methodology available yet that incorporates software reuse as an integral part of the approach. Object-oriented design is a promising approach, but isn't yet fully mature. And while considerable attention is being given to reuse of products throughout the life cycle, no effective, general means have been found to automate the process of identifying available components from statements of needs/requirements. Reuse is also often prevented by the diversity of programming languages in use.

While there are substantial technical issues remaining, the main obstacles to software reuse at present are managerial rather than technical. The Magnavox effort (Carstensen 1987) illustrates that with management determination to reuse software, and with realistic goals and willingness to assume some risks, success can occur even with the use of ad hoc approaches. An important management issue is retaining personnel for "reuse" within

an application area. Other issues are project funding approaches, royalties/licensing fees, financial incentives to a contractor to reuse software rather than create custom software, tax considerations, and copyright laws. The "Not Invented Here" (NIH) syndrome is also an impediment. Effective cost modeling is needed, to analyze potential opportunities to reuse software and to develop software for reuse.

3.2 Recommendations

By all means, software reuse research should continue. The potential payoff is absolutely phenomenal. And, experimentation in large application-oriented domains will provide important insights. The following paragraphs offer some specific suggestions for software reuse research and practice.

Management within organizations should take the initiative to make software reuse a reality. This means much more than just issuing an edict that software reuse will occur--it means setting up workable approaches, with a substantial library of components, and with sufficient support staff. It means committing necessary funds (consistently) to analyze application domains, and identify software that can be effectively reused (from all life-cycle phases). It means spending money "up front", for later gains. If it is to work, positive rewards must come to the

technical personnel involved, who achieve productive reuse. This can both motivate reuse efforts, and help with personnel retention. Management must be prepared to accept some risks in committing to reuse. And, software development and maintenance must have component reuse built in as an integral part of the processes. Perhaps for the near term it isn't as important what the approach is, per se, as that a specific approach is selected and carried out with the collective committment of participating personnel to make it a success. Great emphasis must be placed on the quality of software developed in-house and that brought in for reuse, to build up trust on the part of technical staff; this can alleviate the NIH problem. Realistic reuse guidelines (managerial and technical) must be developed and enforced, emphasizing tailoring to an organization's requirements and standards. Management can solve the issue of language proliferation, by enforcing the use of a single language (e.g., Ada).

Studies should be made of how to effectively share software among companies and the government. This involves legal and economic issues, such as how to compensate a company over time for the investment made in developing software, and how to compensate a contractor for reusing available software rather than developing custom software. The implications of existing government standards for software development (e.g., DoD Std. 2167A) must also be

dealt with, as well as specific mechanisms for government/ contractor contractual interrelationships.

Research should continue in technical approaches as well. There are numerous software development methodologies in place which do not incorporate reuse (e.g., Structured Analysis/Structured Design, SADT). The object-oriented approach holds promise for reuse, due to the direct relationship between components and domain entities. This area needs intensive research; while ad hoc approaches to reuse are better than none, most existing methodologies ignore reuse, when what is needed is encouragement of reuse by the methodologies. A methodology needs to emphasize the awareness of the existence of available software components in the software development and maintenance processes, and accommodation of a design to use available components where practicable. CASE support tools should emphasize specific methodologies, and directly incorporate reuse.

A technical research area with very significant potential, and which should receive immediate emphasis, is that of devising requirements languages whose primitives are tailorable to given application domains. This could permit identification of available components for reuse--with user guidance as necessary. Perhaps a natural approach would be to tie such requirements expression to the object-oriented approach. Underlying this capability, and others discussed in this section, is the requirement for a means for effective domain analysis. Further research into effective

domain analysis methods must also be performed.

The need for help in adapting components is an important near-term goal. There are several important research issues relating to it, that should be emphasized. They include the ideas of a simplified mental model (user model) of systems/components, hypertext, ERA and other library organizations, reverse engineering approaches, module interconnection languages, and code templates. Other approaches for emphasis include automatic component adaptation (parametric approaches, transformation methods, generators, rule-based methods). Emphasis should also be placed on capturing and retaining knowledge about the tradeoffs and decisions made during the life-cycle refinement process, organized for examination and understanding.

Information retrieval (IR) methods are promising relative to reuse retrieval needs, due to their greatly improved execution performance based on emerging special-purpose hardware. The issues of user feedback, and recall, precision and ranking, should be investigated relative to IR use, along with mechanisms for providing sufficient descriptive information to achieve effective identification of candidate components. Examination of effective browsing mechanisms should also be a part of this research.

Careful study should be made of successes in software reuse, to determine what the causes and effects are. Facets

to consider, among others, are: management approaches (organization for reuse, funding approaches, personnel communication), life-cycle methodology, standardization of approaches and language, repository mechanisms, domain analysis methods, "narrowness" of the application domain, experience and quality of the technical personnel. Projects in the U.S. and abroad (especially Britain and Japan) should be analyzed.

Organized, coordinated research and experimentation will give far greater leverage than independent efforts with inevitable overlaps and gaps. The major research projects reviewed in this report all have strengths and shortcomings. Hopefully greater coordination can occur in the future, especially among government-sponsored projects, to the end of a "whole greater than the sum of the parts". The technical research is "getting there" rapidly, with effective solutions to reuse issues being developed. There are some remaining technical issues, and more breakthroughs coming. But there is no technical reason for organizations to wait to implement software reuse. It has proven true in a number of experiments in reuse that, with a concerted management effort and "getting everyone on the team", leverage can be gained quickly from reuse. But it won't "just happen"--it must occur through determined organizational effort. An organization already practicing reuse will be well-positioned to take advantage of more effective methodologies and tools as they appear.

REFERENCES

ANCOAT. 1988. PROCEEDINGS OF THE SIXTH NATIONAL CONFERENCE ON ADA TECHNOLOGY. Arlington, VA (March).

Anderson, C.M. and D.G. McNicholl. 1985. Reusable Software--A Mission Critical Case Study. In Grabow 1985a, p. 205.

Asdjodi, M. 1988. KNOWLEDGE-BASED COMPONENT COMPOSITION: AN APPROACH TO SOFTWARE REUSABILITY. Ph.D. Dissertation, The University of Alabama in Huntsville, Huntsville, AL.

Barnes, B., T. Durek, J. Gaffney and A. Pyster. 1987. Cost Models for Software Reuse. In PROCEEDINGS OF THE TENTH MINNOWBROOK WORKSHOP (1987, SOFTWARE REUSE). Blue Mountain Lake, NY (July).

Barsotti, G. and M. Wilkinson. 1987. Reuseability--Not an Isolated Goal. In Yourdon 1987,pp. A1-A14.

Basili, V.R., H.D. Rombach, J. Bailey, A. Delis and F. Farhat. 1988. Ada Reuse Metrics. In Leslie et al 1988, pp. 11-29.

Bein, J., P. Drew and R. King. 1988. Object-Oriented Data Base Tools to Support Software Engineering. In Leslie et al 1988, pp. 95-110.

Biggerstaff, T.J. and A.J. Perlis (eds.). 1984. Special Issue on Software Reusability. IEEE TRANS. ON SOFTWARE ENGR., vol. SE-10, no. 5 (Sept.).

Biggerstaff, T. and C. Richter. 1987. Reusability Framework, Assessment, and Directions. IEEE SOFTWARE, vol. 4, no. 2 (March), pp. 41-49.

Booch, G. 1987. SOFTWARE COMPONENTS WITH ADA. Benjamin /Cummings, Menlo Park, CA.

Booch, G. and L. Williams (eds.). 1987. PROCEEDINGS OF THE WORKSHOP ON SOFTWARE REUSE (Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium). Boulder, Colorado (October).

Bott, M.F. and P.J.L. Wallis. 1988. Ada and Software Re-use. In Hall 1988a, pp. 177-183.

Brooks, F.P. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE COMPUTER, vol. 20, no. 4 (April), pp. 10-19.

Bullard, C.K., D.S. Guindi, W.B. Ligon, W.M. McCracken, S. Rugaber. 1988. Verification and Validation of Reusable Ada Components. In Leslie et al 1988, pp. 31-53.

Burton, B.A., R.W. Aragon, S.A. Bailey, K.D.Koehler, and L.A. Mayes. 1987. The Reusable Software Library. In Tracz 1987b, pp. 25-33.

Carstensen, H.B. Jr. 1987. A Real Example of Reusing Ada Software. In Yourdon 1987, pp. B1-B19.

Cheatham, T.E., Jr. 1984. Reusability Through Program Transformations. In Biggerstaff and Perlis 1984, pp.589-594.

Chen, Y.F. and C.V. Ramamoorthy. 1986. The C Information Abstractor. In Davis 1986, pp. 291-298.

Conn, R. 1986. Overview of the DoD Ada Software Repository. DR. DOBB'S JOURNAL (Feb.), pp. 60-61,86-93.

Davis, A. (session chmn.) 1986. Reusability of Program Code, session in PROCEEDINGS OF COMPSAC 86, Chicago (October).

Druffel, L. and B. Meyer (eds.). 1988. PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Singapore (April).

EVB. 1987. Creating Reusable Ada Software. In Yourdon 1987, pp. E1-E58. (EVB Software Engineering, Inc.)

Fairley, R., S. Pfleeger and B. Springsteen. 1989. Incentives for Reuse of Ada Components. In notes of Ada Reuse and Metrics Project Technical Status Review, Fairfax, VA (Feb.).

Finkelstein, A. 1988. Re-use of Formatted Requirements Specifications. In Hall 1988a, pp. 186-197.

Fischer, G. 1987. Cognitive View of Reuse and Redesign. In Tracz 1987b, pp. 60-72.

Frakes, W.B. and B.A. Nejmeh. 1987. Software Reuse Through Information Retrieval. In Shriver and Sprague 1987, pp. 530-535.

Fujino, K. 1987. Software Factory Engineering: Today and Future. In Ramamoorthy and Yeh 1987, pp. 262-270.

Gagliano, R.A., M.D. Fraser, and G.S. Owen. 1988. Guidelines for Reusable Ada Library Tools. In Leslie et al 1988, pp. 79-90.

Gautier, B. 1988. Book Review: Software Components With Ada (by Grady Booch). In Hall 1988a, pp. 184-185.

Geary, K. 1988. The Practicalities of Introducing Large-Scale Software Re-use. In Hall 1988a, pp. 172-176.

Goguen, J.A. 1986. Reusing and Interconnecting Software Components. IEEE COMPUTER, vol. 19, no. 2 (February), pp. 16-28.

Grabow, P.C. (session chmn.) 1985a. Software Reuse: Where Are We Going? Conference Session, documented in PROCEEDINGS OF COMPSAC 85, Chicago (October).

Grabow, P.C. 1985b. Software Reuse: Where Are We Going? In Grabow 1985a, p. 202.

Guerrieri, E. 1988. Searching for Reusable Software Components with the RAPID Center Library System. In ANCOAT 1988, pp. 395-406.

Hall, P.A.V. 1987a. Software Components and Reuse. COMPUTER BULLETIN (December), pp. 14-15,20.

Hall, P.A.V. 1987b. Software Components and Reuse--Getting More Out of Your Code. INFORMATION AND SOFTWARE TECHNOLOGY, vol. 29, no. 1 (January), pp. 38-43.

Hall, P.A.V. (guest ed.) 1988a. Software Components and Re-use: special section of SOFTWARE ENGINEERING JOURNAL, vol. 3, no. 5 (September).

Hall, P.A.V. 1988b. Software Components and Re-use. In Hall 1988a, p. 171.

Hocking, D.E. 1988. The Next Level of Reuse. In ANCOAT 1988, pp. 407-410.

Hooper, J.W. 1988. Simulation Model Reuse: Issues and Approaches. In PROCEEDINGS OF THE 1988 SUMMER COMPUTER SIMULATION CONFERENCE, Seattle (July), pp. 51-56.

Horowitz, E. and J.B. Munson. 1984. An Expansive View of Reusable Software. In Biggerstaff and Perlis 1984, pp. 477-487.

Huang, C. 1985. Reusable Software Implementation Technology: A Review of Current Practice. In Grabow 1985a, p. 207.

Hutchinson, J.W. and P.G. Hindley. 1988. A Preliminary Study of Large-Scale Software Re-use. In Hall 1988a, pp. 208-212.

Jones, A., R.E. Bozeman, and W. McIver. 1988. A Framework for Library and Configuration Management. In Leslie et al 1988, pp. 63-78.

Jones, T.C. 1984. Reusability in Programming: A Survey of the State of the Art. In Biggerstaff and Perlis 1984, pp. 488-494.

Kaiser, G.E. and D. Garlan. 1987. Melding Software Systems from Reusable Building Blocks. In Tracz 1987b, pp. 17-24.

King, R. 1988. Object-Oriented Data Base Modeling and Software Environments. In Leslie et al 1988, pp. 91-94.

Lanergan, R.G. and C.A. Grasso. 1987. Software Engineering with Reusable Design and Code. In Biggerstaff and Perlis 1984, pp. 498-501.

Lesslie, P.A., R.O. Chester and M.F. Theofanos. 1988. GUIDELINES DOCUMENT FOR ADA REUSE AND METRICS (DRAFT). Martin Marietta Energy Systems, Inc., Oak Ridge, Tennessee (under contract to U.S. Army AIRMICS).

Lieberherr, K.J. and A.J. Riel. 1988. Demeter: a Case Study of Software Growth Through Parameterized Classes. In Druffel and Meyer, 1988, pp. 254-264.

Machida, S. 1985. Approaches to Software Reusability in Telecommunications Software System. In Grabow 1985a p. 206.

McKay, C.W. 1988. Conceptual and Implementation Models. In Leslie et al 1988, pp. 111-148.

McNicholl, D.G., C. Palmer, et al. 1986. COMMON ADA MISSILE PACKAGES (CAMP). Vol. I: Overview and Commonality Study Results. AFATL-TR-85-93. McDonnell Douglas, St. Louis, MO.

Meyer, B. 1987. Reusability: The Case for Object-Oriented Design. IEEE SOFTWARE, vol. 4, no. 2 (March), pp. 50-64.

Mittermeir, R.T. and W. Rossak. 1987. Software Bases and Software Archives: Alternatives to Support Software Reuse. In Ramamoorthy and Yeh 1987, pp. 21-28.

Murine, G.E. 1987. Recent Japanese Advances in Reusability and Maintainability. In Yourdon 1987, pp. I1-I15.

Neighbors, J.M. 1984. The Draco Approach to Constructing Software from Reusable Components. In Biggerstaff and Perlis 1984, pp. 564-574.

Notkin, D. and W.G. Griswold. 1988. Extension and Software Development. In Druffel and Meyer 1988, pp. 274-283.

Onuegbe, E.O. 1987. Software Classification as an Aid to Reuse: Initial Use as Part of a Rapid Prototyping System. In Shriver and Sprague 1987, pp. 521-529.

Prieto-Diaz, R. 1987. Domain Analysis for Reusability. In PROCEEDINGS OF COMPSAC 87, Tokyo (October), pp. 23-29.

Prieto-Diaz, R. and P. Freeman. 1987. Classifying Software for Reusability. IEEE SOFTWARE, vol. 4, no. 1(Jan.),pp.6-16.

Pyster, A. and B. Barnes. 1987. THE SOFTWARE PRODUCTIVITY CONSORTIUM REUSE PROGRAM. SPC-TN-87-016, December 1987; Software Productivity Consortium, Reston, VA.

Ramamoorthy, C.V., V. Garg, and A. Prakash. 1986. Support for Reusability in Genesis. In Davis 1986, pp. 299-305.

Ramamoorthy, C.V. and R.T. Yeh (eds.) 1987. PROCEEDINGS OF THE 1987 FALL JOINT COMPUTER CONFERENCE, Dallas (October).

Rogerson, A.M. and S.C. Bailin. 1987. Software Reusability Environment Prototype: Experimental Approach. In PROCEEDINGS OF THE TENTH MINNOWBROOK WORKSHOP (1987, SOFTWARE REUSE). Blue Mountain Lake, NY (July).

Sayani, H. 1987. Applications in Reverse Software Engineering. In Yourdon 1987, pp. L1-L15.

Shriver, B.D. and R.H. Sprague, Jr. (eds.). 1987. PROCEEDINGS OF THE TWENTIETH HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, Kailua-Kona, Hawaii (Jan.).

Smith, S.R. 1989. AN ADVANCED FULL-TEXT INFORMATION RETRIEVAL SYSTEM. Ph.D. Dissertation, The University of Alabama in Huntsville, Huntsville, AL.

SofTech. 1985. ISEC REUSABIITY GUIDELINES, December 1985. SofTech, Inc., Waltham, MA.

St. Dennis, R.J. 1987. Reusable Ada Software Guidelines. In Shriver and Sprague 1987, pp. 513-520.

Standish, T.A. 1984. An Essay on Software Reuse. In Biggerstaff and Perlis 1984, pp. 494-497.

Tarumi, H., K. Agusa, and Y. Ohno. 1988. A Programming Environment Supporting Reuse of Object-Oriented Software. In Druffel and Meyer 1988, pp. 265-273.

Tracz, W. 1987a. RECIPE: A Reusable Software Paradigm. In Shriver and Sprague 1987, pp. 546-555.

Tracz, W. (ed.). 1987b. Special Edition, Making Reuse a Reality, IEEE SOFTWARE, vol. 4, no. 4 (July).

Utter, D.F. 1985. Reusable Software Requirements Documents. In Grabow 1985a, pp. 204.

Wood, M. and I. Sommerville. 1988. An Information Retrieval System for Software Components. In Hall 1988a, pp. 198-207.

Wong, W. 1986. A Management Overview of Software Reuse. NBS, Publ. 500-142. Washington, DC.

Yamamoto, S. and S. Isoda. 1986. SOFTDA--A Reuse-Oriented Software Design System. In Davis 1986, pp. 284-290.

Yeh. R.T. and T.A. Welch. 1987. Software Evolution: Forging a Paradigm. In Ramamoorthy and Yeh 1987, pp. 10-12.

Yourdon, E. (ed.). 1987. PROCEEDINGS OF THE CONFERENCE ON SOFTWARE REUSEABILITY AND MAINTAINABILITY (The National Institute for Software Quality and Productivity, Inc.). Tysons, Corner, VA (March).